



ATTORNEY DOCKET NO. 3330/55  
(LOT9-2001-0040)

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of : Glen Salmon et al.  
Serial No. : 09/943,673 Examiner: Qing Yuan Wu  
Filed : August 31, 2001 Group Art Unit: 2126  
Title : THREAD CONSISTENCY SUPPORT SYSTEM AND METHOD

Commissioner for Patents  
P.O. Box 1450  
Alexandria, VA 22313-1450

**DECLARATION OF JOHN KILROY  
UNDER 37 C.F.R. § 1.131**

Sir:

JOHN KILROY hereby declares under penalty of perjury as follows:

1. I am one of the two named inventors in the above referenced U.S. patent application. I am a joint inventor of the subject matter claimed in pending claims 1-11 and 13-33. The other joint inventor of the subject matter claimed in these claims is Glen Salmon. I make this Declaration in support of the patentability of the claims pending in this application.

2. In an Office Action mailed June 22, 2005, claims 1-11 and 13-33 were rejected under 35 U.S.C. § 103(a) as being obvious over U.S. Patent No. 6,801,919 to Hunt et al. ("Hunt"). Hunt was filed on July 27, 2001. The only basis for which Hunt could be considered prior art is under 35 U.S.C. §102(a) or 35 U.S.C. §102(e).

3. Sometime prior to July 27, 2001, I and the other joint inventor conceived and reduced to practice the invention claimed in pending claims 1-11 and 13-33.

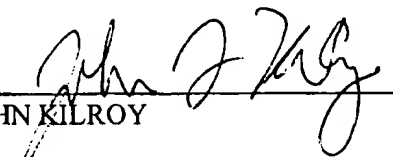
4. Sometime prior to July 27, 2001, I and the other joint inventor provided a description of the subject matter covered by claims 1-11 and 13-33 for inclusion in documents entitled "Lotus Connector API Thread Connection Code" and "Draft Disclosure for the Thread Connection Code". Copies of redacted versions of these documents are attached hereto as Exhibit 1.

5. All of the acts relied upon herein to establish my conception and reduction to practice of the invention were carried out in the United States.

6. Since I and the other joint inventor conceived of and reduced to practice the invention claimed in claims 1-11 and 13-33 of the above referenced application before July 27, 2001, rejection of all these claims over Hunt has been overcome pursuant to Rule 131.

7. All statements made herein of my own knowledge are true and all statements made on information and belief are believed to be true. These statements were made with the knowledge that willful false statements and the like are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and willful, false statements may jeopardize the validity of the application or any patent issuing thereon.

Dated: 11/22, 2005

  
\_\_\_\_\_  
JOHN KILROY

# Lotus Connector API Thread Connection Code

## Release 1 Specification, Draft 0.3

### Introduction

This document is the specification of the Thread Connection Code enhancement to the Lotus Connector API to provide alternate thread support for certain types of Lotus Domino Connectors are used.

The Lotus Connector API provides support for single thread and multi-thread Lotus Domino Connectors. The API assumes connections may be used freely between threads and that each connector supports thread execution without regard for thread consistency. Thread consistency is the need for all related operations to be performed by the same thread. It implies the associated connector will receive the same thread for initialization, get/set properties, Connect, Select, Fetch, Insert, etc. and finally Disconnect. Multiple threads may exist in a thread consistent environment provided each connection is associated with only one thread.

### Audience

The intended audience is technical staff responsible for the development and maintenance of the Lotus Connector API and developers of Lotus Domino Connectors which require thread consistency. The documentation assumes familiarity with the Lotus Connector API and the concept of connectors and metaconnectors. See the LC API Documentation for details of the LC API and developing Lotus Connectors and MetaConnectors.

When reading the document please note the following:

Gray text denotes function that will be delivered later than the first release. By implication, all other text denotes function that will be delivered in the first release. An example of a feature not implemented in the first release would include additional color markup.

Green text marked with revision bars denotes changes from the first draft of the specification. Underline implies new text and strikethrough represents removed text. An example of a revision would include this notation.

A glossary of terms and abbreviations is included in Appendix A.

### Objectives

The objectives of this document are as follows:

- To specify the attributes and behavior of Lotus Domino Connectors requiring thread consistency and the Thread Connection Code which provides thread consistency to such connectors.

- To specify the changes required to the Lotus Connector API resulting from the first objective.

### Overview

The following diagrams gives a high level overview of the thread consistency support provided by the Thread Connection Code. The example depicted in the next two diagrams is of an application which has four threads. Three threads share a connection and perform a number of data operations. The remaining thread has a separate connection and performs all of its own data operations. The first diagram represents the Lotus Connector API conventional thread support. The second diagram represents the API with Thread Connection Code to maintain thread consistency.

## Lotus Connector API Conventional Thread Support

In the example application, the first thread establishes a new connection, initializes it and connects, then selects and fetches data. A second thread is created which uses the existing connection to select, fetch, and then update data. A third

thread establishes a new connection, initializes and connects and then selects and fetches data. Later it updates the data and then disconnects and terminates the connection. While the third thread is still active, a fourth thread uses the first connection to remove data. In a more detailed example, throughout this process, the individual threads may perform additional data operations.

## Lotus Connector API with Thread Connection Code

In the example application, the first thread establishes a new connection. The Thread Connection Code isolates the connection from the requesting thread. It creates its own thread and uses it to create the new connection. The TCC then maintains a correlation between its thread and the connection handle. The application's first thread initializes and connects to the connection. Each call to the connection goes through the TCC which takes the connection handle and uses it to find its internal thread which is dedicated to the connection. The application thread selects and fetches data. Again, the TCC uses the connection handle to identify its internal thread and uses the internal thread when interacting with the connection. A second thread is created which uses the existing connection to select, fetch, and then update data. The TCC identifies its first thread as matching the connection handle. A third application thread establishes a new connection. The TCC creates another internal thread to correspond to the new connection. The application's third thread initializes and connects and then selects and fetches data. The TCC identifies its second internal thread as matching the connection handle which is then used to complete the data access. A fourth application thread uses the first connection to remove data. The TCC, using the connection handle identifies its first thread to complete the operation. Later, the application's third thread updates and then disconnects and terminates the connection. The TCC identifies its second internal thread as corresponding to the connection handle and then uses it. In a more detailed example, throughout this process, the individual application threads may perform additional data operations.

### Introduction

The Thread Connection Code provides an interface layer between an application written to the Lotus Connector API and an individual Lotus Domino Connector. The TCC is only active for connectors which require thread consistency. Connectors requiring TCC support will indicate so as part of their identification. When a connector identifies the need for this special thread support, the LC API automatically activates the TCC to isolate applications threads from the threads running through the connector.

### Supported Platforms and Products

The Thread Connection Code is supported on the following platforms:

- Windows NT 4.0
- Windows 95
- Windows 98
- OS/2 Warp 4.0
- AIX 4.3.1
- Solaris SPARC 2.5
- HP-UX 11.0
- Linux
- AS/400
- S/390

### Thread Connection Code Architecture

The Thread Connection Code or TCC, implements a level of indirection between an application's threads to one or more connections and internal threads, mapping to preserve one thread per connection. the TCC is broken into two components, the detection / activation code and the The Threading MetaConnector or TMC. The detection and activation code is an enhancement to the LCConnectionAlloc() function of the LC API while the TMC is entirely new code. Since the LC API is written in "C", the detection and activation code must be written in "C" for consistency. However, there is no language restriction for the TMC which lends itself to both "C" and "C++".

## Part 1: LC API Detection and Activation code

The LC API loads a connector as part of the `LCConnectionAlloc()` routine. At load time, the connector's `LCXIdentify()` function is called to gather information about the connector and what support it provides. The connector's thread support is part of the identification data.

A new flag, `LCIDENTIFYF_CONTEXT_THREAD`, within the `lcIdentifyFlags` entry of the `LCXIDENTIFY` identification structure, indicates the connector requires special handling of threads. This type of connector is known as a Thread Dependent Connector or TDC. The flag indicates the connector requires thread consistency, i.e. the same thread which initializes a connection must be used for all subsequent operations on that connection. It does not indicate the connector is single threaded. Multiple threads may exist when multiple connections are active at one time.

The Thread Connection Code provides special thread handling support to the Thread Dependent Connector. The TCC is implemented in two parts. Within the LC API exists a small amount of code which performs detection of TDCs. Once a TDC is detected, the code loads and activates the Threading MetaConnector or TMC, which performs the thread isolation code.

When an application calls `LCConnectionAlloc()` to load a connector, the Thread Connection Code tests the connector's thread information. If the `LCIDENTIFYF_CONTEXT_THREAD` flag is present, the TCC calls `LCConnectionAlloc()` to load and initialize the Threading MetaConnector. Once the TMC has been created, the original connection is assigned as the TMC's subconnection. The connection handle for the TMC is returned to the calling application in place of the requested connection handle.

The changes for `LCConnectionAlloc()` are outlined in the following pseudo code:

```
LCConnectionAlloc (... ConnectorName ...)
...
Load and Initialize newConnection
...
if (Successful)
    if (newConnection -> lcIdentifyFlags has LCIDENTIFYF_CONTEXT_THREAD flag)
        LCConnectionAlloc (... 'TMC' ...)
        LCConnectionSetProperty (TMC, LCTOKENF_SUBCONNECTOR, newConnection)
        return TMC handle
    else
        return newConnection handle
    end_if
end_if
end_sub
```

Note: To maintain transparency, the TMC reports back the subconnector's responses (those of the TDC) for all properties including `LCTOKEN_CONNECTOR_NAME`. Additionally, the LCX Methods appear as those of the underlying subconnector.

## Part 2: Threading MetaConnector code

The Threading MetaConnector receives calls from the LC API and pass those calls on to the Thread Dependent Connector. The LC API may receive calls from multiple threads to a single connection where as the TMC creates and uses one thread for the duration of each connection. As with all connectors and metaconnectors, the TMC creates a context object when a new connection is initialized. As part of the context object, the TMC maintains both the handle of the TDC and a Thread Support Object or TSO.

The TSO consists of two semaphores, a data passing structure, and a thread executing within a message processing routine. In this implementation, the TCC uses threading in conjunction with a toggle algorithm to isolate thread execution. By utilizing two semaphores, execution toggles between the thread executing from the application and the TSO's thread.

The following pseudo code outlines the execution toggling between the application code and the TSO. LCXInitialize and LCXTerminate are responsible for configuration and clean up of the Thread Support Object. The remaining LCX methods, with the exception of LCXIdentify, are processed through the TSO

### Application Code thread

running from the application through the LC API  
and into the Threading MetaConnector

```
(LCXInitialize)
Rem Semaphores are 'per context block' ...
Rem    and used for handshaking

Create and Open Semaphore #1 and #2

Request Semaphore #1
Start Thread Support Object, passing the ...
address of the context block as the
parameter
```

ready for LCX Methods calls

```
(LCX Method)
set message ID for LCX Method in context
block
Load method specific parameters ...
into the context block
Release Semaphore #1
Request Semaphore #2
```

### Thread Support Object thread

running inside the TSO processing routine

```
Initialize
Open Semaphore #1 and #2

Request Semaphore #2
```

ready for LCX Methods processing

```
Loop
Request Semaphore #1

get message ID from context block

switch on ID
  TSOP_GETPROPERTY_ID:
    get method parameters specific to ...
    "GetProperty" from context block
    execute LCXGetProperty for the TDC
    store results from subconnector ...
    in context block

  TSOP_SETPROPERTY_ID:
    . . .
  TSOP_CONNECT_ID:
    . . . etc . . .
end_switch
Release Semaphore #2
if (ID is TSOP_TERMINATE_ID)
  perform TSP cleanup and termination
  Close Semaphore #1 and #2
  exit_loop
end_if
```

get method specific results from context block

finished all LCX Methods calls

```
(LCXTerminate)
. complete as with other LCX Methods ...

Close Semaphore #1 & #2
Destroy Semaphore #1 & #2
```

The Threading MetaConnector's context block contains elements for the Thread Dependent Connector's connection handle as well as the data passing structure. A pointer to the block is passed when the Thread Support Object's thread is first created on the processing routine.

The data passing structure contains the union of all parameters which are used as input and output from all LCX methods of the connector API. The following is a list of the parameters and the LCX methods which use each.

### Methods Required by the LC API for Connectors and Metaconnectors

There are 17 methods required by the LC API for any connector or metaconnector. Not all methods are supported by all connectors however each must be implemented and may return LCFIL\_UNAVAILABLE when not supported.

The Threading MetaConnector has specific code associated with initialization and termination. For all remaining functional methods, the TMC performs parameter loading and unloading and thread toggling. The individual methods are outlined in the following section with details for specific methods as well the generic methods processing.

### LCXIdentify

This is the first method called when the Threading MetaConnector is loaded. It returns the metaconnector's LCXIDENTIFY structure.

The LCXIDENTIFY structure indicates levels of support as well as the size of the context block required. The context block contains internal data elements as well as the parameter passing structure for the Thread Support Object for the LCX Methods. The following is the minimal data structures for the TSO parameters and the context block. Additional data elements may be added as needed.

```
typedef struct {
    LCSTATUS      lcStatus;
    LCCONNECTION  lcSubConnection;
    LCINT         lcActionType;
    LCFIELDLIST   lcParmFieldlist;
    LCINT         lcRecordIndex;
    LCFIELDLIST   lcDestFieldlist;
    LCINT*        lcAffectedCount;
    LCINT         lcObjectType;
    LCFIELDLIST   lcSrcFieldlist;
    LCSTREAM*     lcStatement;
    LCSINT        lcRecordCount;
    LCINT         lcPropertyToken;
    LCFIELD       lcDestField;
```

```

LCFIELDLIST    lcKeyFieldlist;
LCFIELD        lcSrcField;

} TSOPARMS;

typedef struct {
LCCRITSEC      lcMethodCritSec;
LCCRITSEC      lcProcessingSec;
LCCRITSEC      lcFinishedCritSec;
LCINT          lcMethodID; /* indicates which LCX Method to process */
TSOPARMS       lcTSOParms;

} TSO;

typedef struct {
LCCONNECTION   lcThisConnection;
LCSTREAM       stream; /* optional temporary internal usage stream */
TSO            tso;
} METACONTEXT;

```

### LCXInitialize

This method performs the internal initialization specific to the current thread, including initialization of the context block. If the LCINITTERM\_PROCESS flag is set, this method additionally performs metaconnector initialization specific to the whole process.

The context block is initialized. An optional general purpose stream object may be part of the context block. If so, it is cleared. Note: Additional objects may be stored in the context block for optimization purposes - any such objects would be initialized at this time. As part of the context block, the TSO is initialized including the parameter structure and the semaphores, necessary for toggling execution of the TSO processing routine. Additionally, the TSO thread is spawned on its processing routine. The LC API function is as follows:

```
status = LCProcThreadCreate (TSOProcessingRoutine, (void *)context_block, &pNewThread);
```

The processing routine initializes any internal data and then waits for processing requests.

Suggestion: The most efficient method of initializing the context block is often to first use memset() to initialize all bytes to zero and then setting any nonzero elements to their initial values.

### LCXTerminate

This method performs the termination specific to the current thread. If the LCINITTERM\_PROCESS flag is set, this method additionally performs the termination specific to the whole process.

The lcMethodID of the Thread Support Object is set to TSOP\_TERMINATE\_ID and the input parameters are loaded into the lcTOSPParms structure of the context block for the Thread Dependent Connector's LCXTerminate. In addition to calling the LCConnectionTerminate() with the Thread Dependent Connector handle, the TSO performs its internal cleanup and then exits the process routine, thereby terminating the TSO thread. The Threading MetaConnector cleans up the TSO including closing the semaphores. If the context block contains additional objects such as a general purpose stream or allocated any buffers, these are freed and cleaned up here.

### LCX Methods

For all remaining m...

Conversion terminated at this point (TRIAL version of software).

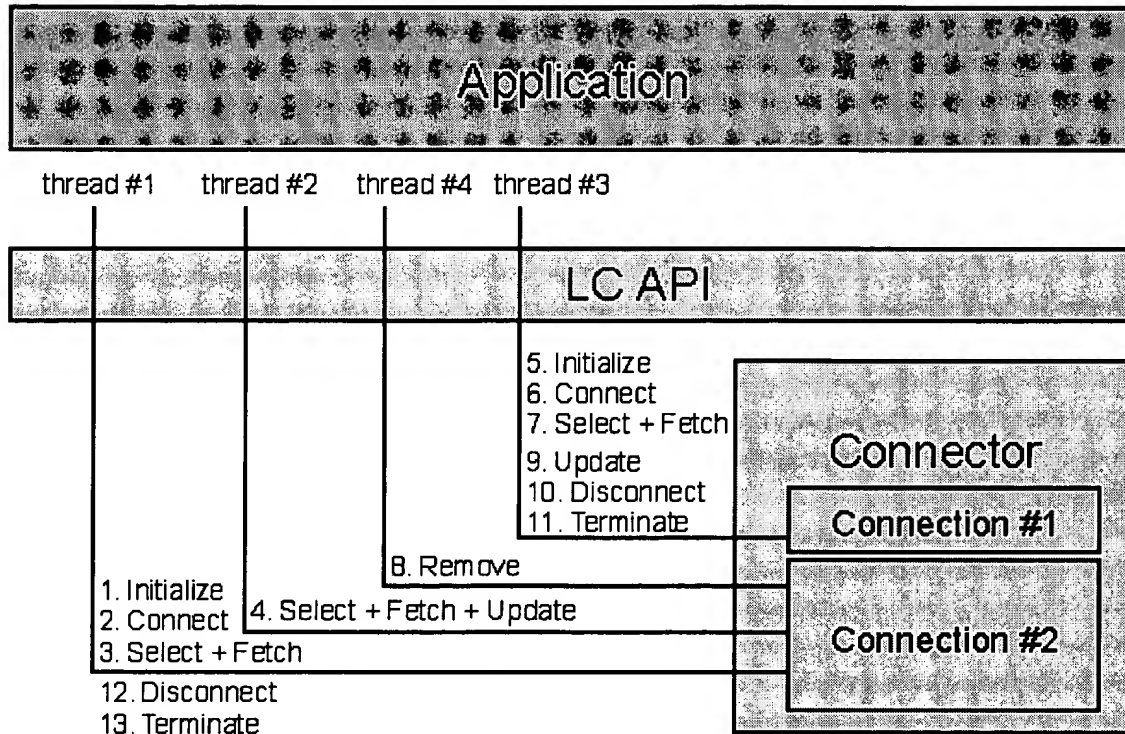


## Draft Disclosure for the Thread Connection Code

The thread-connection code allows any thread from a multi-threaded server application to successfully perform operations against a restrictive back-end system that confines operations to a particular thread, usually, the thread to which the back-end has granted a connection handle. The thread-connection code is a cross-platform solution which employs its own abstracted versions of semaphore and thread primitives that are available on Windows and Unix platforms to maintain a consistent relationship between a requesting thread and a connection handle. A test harness along with validation functionality incorporated within the thread-connection code allows verification of proper system behavior.

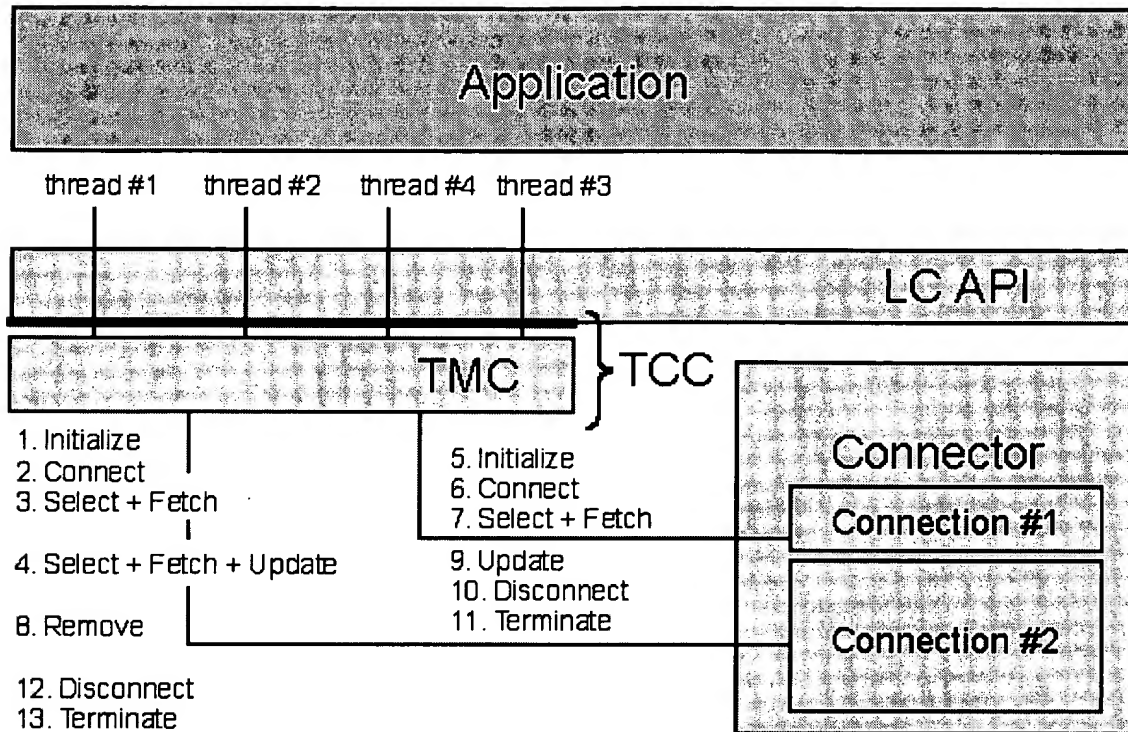
The solution allows a multi-threaded, server-based system, like Domino, to interoperate with a restrictive back-end system without requiring any modification.

The following diagrams gives a high level overview of the thread consistency support provided by the Thread Connection Code. The example depicted in the next two diagrams is of an application which has four threads. Three threads share a connection and perform a number of data operations. The remaining thread has a separate connection and performs all of its own data operations. The first diagram represents the Lotus Connector API conventional thread support. The second diagram represents the API with Thread Connection Code to maintain thread consistency.



#### Lotus Connector API Conventional Thread Support

In the example application, the first thread establishes a new connection, initializes it and connects, then selects and fetches data. A second thread is created which uses the existing connection to select, fetch, and then update data. A third thread establishes a new connection, initializes and connects and then selects and fetches data. Later it updates the data and then disconnects and terminates the connection. While the third thread is still active, a fourth thread uses the first connection to remove data. In a more detailed example, throughout this process, the individual threads may perform additional data operations.



Lotus Connector API with Thread Connection Code

In the example application, the first thread establishes a new connection. The Thread Connection Code isolates the connection from the requesting thread. It creates its own thread and uses it to create the new connection. The TCC then maintains a correlation between its thread and the connection handle. The application's first thread initializes and connects to the connection. Each call to the connection goes through the TCC which takes the connection handle and uses it to find its internal thread which is dedicated to the connection. The application thread selects and fetches data. Again, the TCC uses the connection handle to identify its internal thread and uses the internal thread when interacting with the connection. A second thread is created which uses the existing connection to select, fetch, and then update data. The TCC identifies its first thread as matching the connection handle. A third application thread establishes a new connection. The TCC creates another internal thread to correspond to the new connection. The application's third thread initializes and connects and then selects and fetches data. The TCC identifies its second internal thread as matching the connection handle which is then uses to complete the data access. A fourth application thread uses the first connection to remove data. The TCC, using the connection handle identifies its first thread to complete the operation. Later, the application's third thread updates and then disconnects and terminates the connection. The TCC identifies its second internal thread as corresponding to the connection handle and then uses it. In a more detailed example, throughout this process, the individual application threads may perform additional data operations.